ORB in 5ms: An efficient SIMD friendly implementation

Prashanth Viswanath, Pramod Swami, Kumar Desappan, Anshu Jain, Anoop Pathayapurakkal

Texas Instruments India Private Ltd, Bangalore, Karnataka, India

Abstract. One of the key challenges today in computer vision applications is to be able to reliably detect features in real-time. The most prominent feature extraction methods are Speeded up Robust Features(SURF), Scale Invariant Feature Transform(SIFT) and Oriented FAST and Rotated BRIEF(ORB), which have proved to yield reliable features for applications such as object recognition and tracking. In this paper, we propose an efficient single instruction multiple data(SIMD) friendly implementation of ORB. This solution shows that ORB feature extraction can be effectively implemented in about 5.5ms on a Vector SIMD engine such as Embedded Vision Engine(EVE) of Texas Instruments(TI). We also show that our implementation is reliable with the help of repeatability test.

1 Introduction

Keypoint detectors and descriptors play an important role in computer vision applications such as object recognition, image stitching, structure from motion etc. The most commonly used methods are Scale Invariant Feature Transform(SIFT) [1], Speeded up Robust Features(SURF) [2] and Oriented FAST and Rotated BRIEF(ORB) [3]. These methods have been proven to be reliable in detecting features in real world images. However, the biggest challenge is to meet the real-time performance requirements. In most applications, keypoint extraction is followed by computationally intensive processing such as tracking the features or recovering 3D points and so on. Ethan Rublee et al. [3] shows that it takes about 15.3ms to compute ORB descriptors for roughly 1000 keypoints in a 640x480 image on an Intel i7 2.8GHz processor. Kwang-yeob Lee and Kyung-jin Byun [4] proposes a hardware accelerator for ORB whose run time is 18ms for an input image of 640x480. Our solution takes about 5.5ms to process 3 levels of the input image whose base resolution is 400x400 and compute 500 descriptors on Embedded Vision Engine(EVE) of Texas Instruments(TI) running at 650MHz [5].

In this paper, we propose a computationally efficient implementation of ORB which is suited for single instruction multiple data(SIMD) architecture. Our contributions are as follows:

– An alternate SIMD friendly implementation of FAST9 keypoint detection

– An alternate approach to computing FAST9 score instead of the iterative approach keeping the definition of FAST9 score the same, which is the highest threshold at which a keypoint still remains a keypoint [6]
– Sparse point non-maximal suppression based on the FAST9 score
– SIMD friendly implementation of the Harris score and rBRIEF descriptor computation

We also share the performance details of the implementation on EVE which is part of the TDA2x device of TI. EVE is a fully programmable accelerator created specifically to enable the processing, latency and reliability needs found in computer vision applications. The EVE includes one 32-bit Application-Specific RISC Processor(ARP32) and one 512-bit Vector Coprocessor(VCOP) with built-in mechanisms and unique vision-specialized instructions for concurrent, low-overhead processing. The VCOP is a SIMD engine with built-in loop control and address generation. It is a dual 8-way SIMD engine. It has certain special properties such as transpose store, de-interleave load, interleaving store and so on. The VCOP also has specialized pipelines for accelerating lookup tables and histograms [5]. To validate the ORB implementation, we perform the repeatability test for images with varying view points and blurring.

The rest of the paper is organized as follows: Section 2 provides the details of our implementation, Section 3 shows the results and performance of the implementation on EVE and Section 4 offers conclusions as to effectiveness of our implementation.

2 Proposed Solution

2.1 Overview

The ORB algorithm flow is as shown in Fig. 1. The input to our algorithm is an 8-bit image of size WxH. The output is a list of 256-bit ORB descriptor, the corresponding XY co-ordinates and the level of image at which the feature was detected. Since FAST is not a multi-scale algorithm, we obtain different levels of the image using image pyramid and employ FAST9 feature detection on each level. The FAST9 detector outputs the list of XY co-ordinates in 32-bit packed format indicating the location of the keypoints. For these keypoints, we compute the FAST9 score as explained in Section 2.3. We compute FAST9 score only for the keypoints and not every pixel of the image. This results in significant reduction of memory bandwidth and computational requirements. After the FAST9 score computation, we apply non-maximal suppression as detailed in Section 2.4. Since FAST9 keypoints are clustered together, we apply 4-way non-maximal suppression which suppresses non-maximas considering neighbors in four directions: top, bottow, left and right. The traditional approach of non-maximal suppression involves applying a sliding 3x3 window across the entire image and determine the non-maximas. However, our implementation of non-maximal suppression operates on the sparse keypoints directly. This further results in significant reduction of memory bandwidth and computational requirements. After suppression, we sort the non-suppressed keypoints based on the FAST9 score and output the best *2N* keypoints. For these best keypoints, we compute the Harris score. At this stage, we have multiple lists of co-ordinates and their Harris score, each corresponding to a particular image level. We then sort the lists across multiple levels based on the Harris score and output the best N keypoints for which the rBRIEF descriptors are computed.

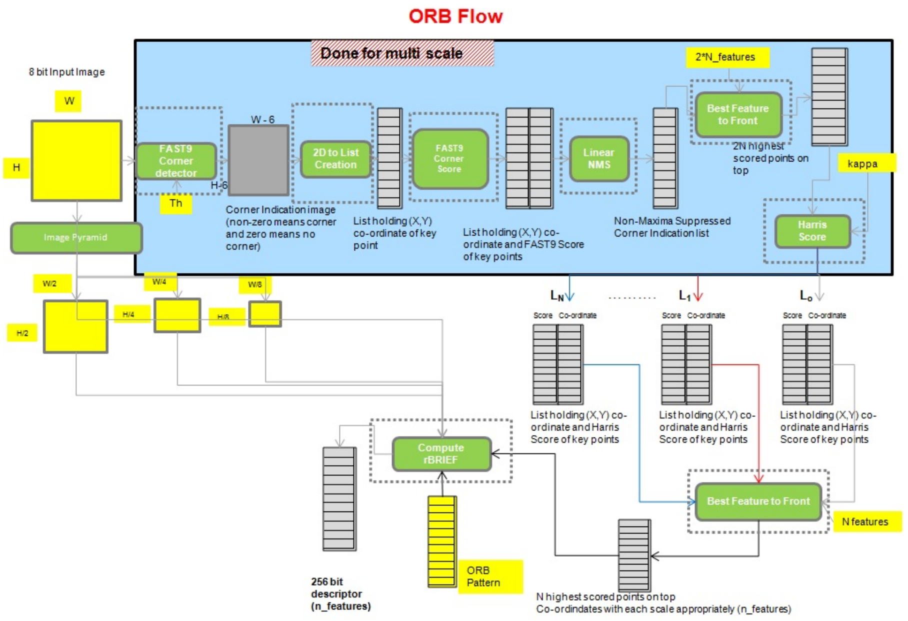

ORB Flow
Done for multi scale
8 bit Input Image
W
H
FAST9 Corner detector
Th
W - 6
H-6
Corner Indication image (non-zero means corner and zero means no corner)
2D to List Creation
List holding (X,Y) co-ordinate of key point
FAST9 Corner Score
List holding (X,Y) co-ordinate and FAST9 Score of key points
Linear NMS
Non-Maxima Suppressed Corner Indication list
2*N_features
Best Feature to Front
2N highest scored points on top
kappa
Harris Score
Image Pyramid
W/2
H/2
W/4
H/4
W/8
H/8
L_N
....
L_1
L_o
Score Co-ordinate
Score Co-ordinate
Score Co-ordinate
List holding (X,Y) co-ordinate and Harris Score of key points
List holding (X,Y) co-ordinate and Harris Score of key points
List holding (X,Y) co-ordinate and Harris Score of key points
Best Feature to Front
N features
Compute rBRIEF
ORB Pattern
256 bit descriptor (n_features)
N highest scored points on top
Co-ordindates with each scale appropriately (n_features)

Fig. 1. ORB algorithm flow.

2.2 FAST9 keypoint detection

FAST is a popular feature detector in real time systems. FAST algorithm picks a 7x7 window around each pixel p as shown in Fig. 2. It takes intensity threshold between the center pixel p and those in a circular ring around p as an input parameter. The algorithm checks if there is a contiguous arc of K or more pixels in the circular ring which satisfy either the bright or dark condition. We use FAST9 (K = 9), which has been shown to have good performance [7] [6].

Although the algorithm is simple, it poses following challenges for a Vector SIMD engine:

– Though the pixel access pattern around each pixel in a 7x7 window is fixed, these are non-sequential locations and hence not friendly towards a simple vector load instruction.

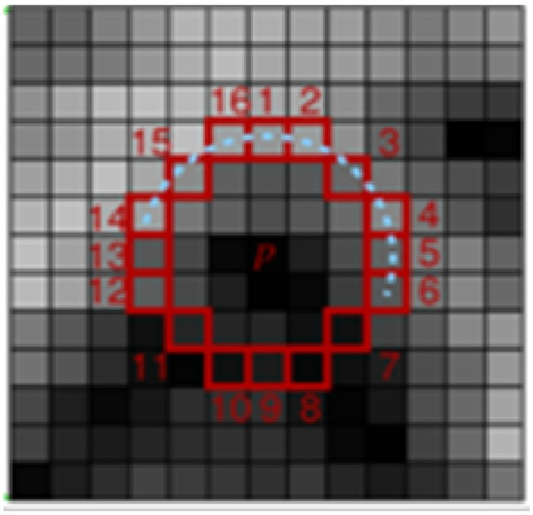


Fig. 2. FAST9 pixel pattern. This image is taken from http://www.edwardrosten.com/work/fast.html.

– For every pixel, we need to check if there is a contiguous arc of K pixels that satisfy the FAST property. This results in the need to check 16+(K-1) combinations. In the case of FAST9, we would need to check 24 combinations.

Traditional approach to check how many consecutive pixels in the circular ring that are similar involves running a loop 16+(K-1) times which updates a counter to indicate the number of pixels satisfying the condition. It also needs to reset the counter selectively for the appropriate elements while maintaining the status of other elements. This level of control logic does not work well in Vector SIMD engines. Hence, we propose a simple 'SHIFT' and 'AND' based technique to find K consecutive pixels which are similar.

We store the comparison of 16 offset pixels with the center pixel in a bit packed format such that we form a 16-bit mask for each center pixel. If bit number 5 is 0, it means that pixel 5 in the circular ring is not similar to center pixel and so on. This kind of mask is simple to generate on a Vector SIMD engine. Since the contiguous arc of K similar pixels can be in any location (i.e last bit followed by first *K-1* bits), we need to do a wrap around search. This can be easily accommodated by duplicating the mask and generate a 32-bit mask. For FAST9, it is sufficient if we duplicate only lower 8 bits of the mask and place it from bit 17-24 to obtain a 24-bit mask X. Now, we can perform the 'SHIFT' and 'AND' logic as follows:

Pseudo code to find 9 consecutive similar pixels

```
begin
// Will tell if there are 2 consecutive similar pixels
```

```
    X1 = X >> 1
    X2 = X1 & X
// Will tell if there are 4 consecutive similar pixels
    X3 = X2 >> 2
    X4 = X3 & X2
// Will tell if there are 8 consecutive similar pixels
    X5 = X4 >> 4
    X6 = X5 & X4
// Will tell if there are 9 consecutive similar pixels
    X7 = X6 >> 1
    X8 = X7 & X6
    If X8 != 0, mark pixel as a corner
end.
```

As is apparent, this technique has logarithmic convergence. This approach requires just 4 steps against the 16 + (9-1) = 24 steps required in the traditional approach for FAST9 keypoint detection. For FAST12, we would also need just 4 steps with the change in the shift factor from 1 to 4 in the last step.

In order to address the first challenge, we use 17 vector load instructions of SIMD width, one vector load for the center pixels, and 16 vector loads for the offset pixels in the circular ring. Although this approach has certain short comings such as minimal data reuse, it is efficient since we are operating on SIMD width elements at a time. In EVE, the SIMD width is 8. Hence, we can work on 8 pixels at a time. Hence, there is an 8x performance benefit.

The ORB requires orientation of FAST keypoints. This is computed at a later stage while computing the descriptor rather than at FAST keypoint detection stage, since the detecting stage operates at every pixel in the image as opposed to the descriptor stage which is operating only on the best N key points.

As mentioned earlier, the input to the FAST9 keypoint detector is an input image or a level of the image. The output is the XY co-ordinate list in 32-bit packed format (16-bit X followed by 16-bit Y) indicating the location of the keypoint. It is important that the XY co-ordinate list is generated in the raster scan order of the image for applying the sparse point non-maximal suppression. This is explained in Section 2.4.

This method of FAST9 detection has been implemented on EVE and the core compute performance is 5.3 cycles/pixel. This implementation has the same cycle count for every pixel as opposed to other approaches such as machine learning approach whose cycle count is highly data dependent [6] [7].

2.3 FAST9 score computation

FAST9 score is defined as the threshold for which, a FAST9 key point still remains a key point [6]. For a given FAST9 key point, its score is given by the highest threshold for which the key point still has 9 contiguous similar pixels around it. The traditional approach of computing the score would involve iteratively incrementing the threshold and check if a key point still remains a key point [6]. The challenges posed by this approach are:

- The keypoints given by FAST9 detector are sparse in nature and hence poses challenges in vectorizing the operations
- The iterative approach of computing the score and conditionally exiting is not suitable for Vector SIMD engines

FAST9 detector gives the XY location of the key points. We pick 7x7 window of pixels around each of the key point. In order to be able to vectorize our computations, we re-order the data by picking only the 16 pixels of the circular ring from the 7x7 window and place them consecutively. Again, in order to take into account the wrap around nature of FAST9, we pick the lower 8 pixels again and place them as well. Hence, for every key point we have 24 offset pixels placed consecutively.

Next, we compute maximum and minimum intensity values of the offset pixels taking them 9 at time, i.e max(0-8), max(1-9)...max(15-23) and min(0-8), min(1-9)..min(15-23). Hence, for every center pixel key point, we have 16 maxima and minima values. Maxima are used if the center pixel satisfies the dark condition(center pixel is darker compared to the offset pixels) of the FAST9 algorithm and Minima are used if the center pixel satisfies the bright condition(center pixel is brighter compared to the offset pixels) of the FAST9 algorithm. Since we do not indicate whether the keypoint is bright or dark during the FAST9 keypoint detection stage, we compute both maxima and minima in this stage. Out of the 16 maxima and minima, only one of them is the score. In order to compute that, we find minimum value *Vmin* from the maxima values and maximum value *Vmax* from the minima values. We compare the *Vmin* and *Vmax* with the center pixel. There are only two possibilities: either *Vmin* and *Vmax* are greater than center pixel intensity or *Vmin* and *Vmax* are lesser than center pixel intensity. The other two possibilities where *Vmin* is greater than center pixel intensity while *Vmax* is lesser than center pixel intensity and vice versa are not possible by the definition of FAST9. Hence, the final score is computed as follows:

- If *Vmax* and *Vmin* are greater than center pixel, the score is the minimum of difference of *Vmax* and *Vmin* with the center pixel intensity minus one. This represents the dark condition.
- If *Vmax* and *Vmin* are lesser than center pixel, the score is the maximum of difference of the center pixel with *Vmax* and *Vmin* minus one. This represents the bright condition.

Pseudo code to compute FAST9 score

```
begin
    cb = center pixel;
    for(startpos=0; startpos<16; startpos++)
    {
        pMin = co[startpos]; // co[] = offset pixel array
        pMax = pMin;
        for(i=1; i<9; i++)
        {
            if(pMax < co[startpos+i])
                pMax = co[startpos+i];
            if(pMin > co[startpos+i])
                pMin = co[startpos+i];
        }
        Bscore[startpos] = pMax; // Bscore[] = array of maxima
        Dscore[startpos] = pMin; // Dscore[] = array of minima
    }
    score_b = Bscore[0];
    score_d = Dscore[0];
    for(i=1; i<16; i++)
    {
        if(score_b > Bscore[i])
            score_b = Bscore[i];
        if(score_d < Dscore[i])
            score_d = Dscore[i];
    }
    if((score_b > cb) && (score_d > cb))
    {
        if(score_b > score_d)
            score = score_d - cb - 1;
        else
            score = score_b - cb - 1;
    }
    else if ((score_b < cb) && (score_d < cb))
    {
        if(score_b > score_d)
            score = cb - score_b - 1;
        else
            score = cb - score_d - 1;
    }
end.
```

The above approach is not iterative and involves simple operation such as max and min which are supported by most of Vector SIMD engines. The core compute performance of this approach is 31.5 cycles/keypoint on the EVE engine of TI. For the data rearrangement, we use the table look up hardware and transpose store property which are supported by EVE [5]. The speed up of computing FAST9 score is due to:

- Vectorizing the operations to compute FAST9 score and non-iterative approach
- Operating on keypoints only and not every pixel of the image

2.4 Sparse point non-maximal suppression

FAST9 keypoints obtained for an image are generally clustered. Hence, non-maximal suppression is an important step to obtain the most reliable feature points. Traditional approach of non-maximal suppression involves applying a 2D 3x3 running window across the input and retain only the maxima in the window. This approach has the following disadvantages:

– FAST9 score has to be computed for every pixel of the input image in order apply the 2D suppression which results in significant wastage of compute cycles and memory bandwidth
– Assuming that FAST9 score is computed only for the keypoints, it has to be mapped back into the 2D image structure which is again a challenging task

Hence we propose an approach which operates on sparse keypoints directly without the 2D notion and can be vectorized easily. This approach is split into two stages: horizontal non-maximal suppression and vertical non-maximal suppression.

Horizontal non-maximal suppression: In this stage, we find the maxima along the X direction. For every keypoint, we check if it has a neighbor in either the left or right direction and then compare their FAST9 scores to suppress the non-maximas. As it was mentioned earlier, it is important that the XY co-ordinate list of keypoints are in raster order. This would mean that the keypoints would be listed in buckets of Y, i.e, same Y but different X (example XY list - 0x0303, 0x0403, 0x0803, 0x0404.. and so on). Once we have the data in this format, we can easily determine if a neighbor exits in the right or left by checking against (X-1,Y) and (X+1,Y). We can also compare the FAST9 scores accordingly and suppress the non-maximum keypoints along the horizontal direction. While storing the co-ordinates of the non-suppressed keypoints, we pack them with ID such that the 32-bit output is 10-bit X, followed by 10-bit Y and 12-bit ID. This ID is used in the next stage of vertical non-maximal suppression. As you can see, this implementation is simple and vector friendly.

Pseudo code to for Horizontal non-maximal suppression

```
begin
// i = 1 to num_corners-1 since we need 1 pixel border in each side
    for(i=1; i<(num_corners-1); i++)
    {
        left_xy = corners[i-1];
        center_xy = corners[i];
        right_xy = corners[i+1];
        left_scr = scores[i-1];
        center_scr = scores[i];
        right_scr = scores[i+1];
        left_xy += 0x10000;
        right_xy -= 0x10000;
//Generate right and left neighbor mask
```

```
        Vnf1 = (left_xy == center_xy);
        Vnf2 = (right_xy == center_xy);
//Generate score mask
        Vsf1 = (center_scr <= left_scr);
        Vsf2 = (center_scr <= right_scr);
        Vf1 = Vnf1 & Vsf1;
        Vf2 = Vnf2 & Vsf2;
//Final mask indicating the neighbor and the maximum score
        Vf1 |= Vf2;
        x = center_xy & 0xFFFF0000;
        y = center_xy & 0x0000FFFF;
        // pack X, Y and ID: 10 bit X, 10 bit Y and 12 bit ID
        nms_x_corners[i] = (x << 6) | (y << 12) | (i);
        if(Vf1)
            nms_x_score[i] = 0;
        else
            nms_x_score[i] = scores[i];
    }
end.
```

Vertical non-maximal suppression: In order to suppress along the Y direction (top and bottom), it would be easy if the co-ordinate list is arranged in raster order of Y, i.e. buckets of X, same X and different Y (example XY list - 0x0303, 0x0304, 0x0306, 0x0403.. and so on). In order to obtain the data in this format, we sort the XY-ID output from the horizontal non-maximal suppression stage in ascending order. Since the XY-ID list is now in different order, we need to obtain the corresponding score values. This is done using the ID to look-up the score values of the corresponding XY. Once the score values are re-ordered, we can follow the same approach as in horizontal suppression to suppress the non-maximas in top and bottom direction.

In order to obtain the best *2N* keypoints, we then sort the non-suppressed keypoints based on the FAST9 score and output it. The horizontal non-maximal suppression and vertical non-maximal suppression takes 1.8 cycles/keypoint and 1.3 cycles/keypoint respectively on EVE. 32-bit sort takes 4.78 cycles/point for a 2048-point sort on EVE. The speed up of is due to:

– Vectorizing the suppression operations
– Operating on sparse keypoints only and not every pixel of the image

2.5 Harris score computation and rBRIEF descriptor computation

Since literature suggests that FAST does not produce a measure of cornerness and has large responses along edges, Harris score is used to order the FAST keypoints [3]. Harris score is computed by picking a 7x7 window around each keypoint and computing the gradient and the tensor matrix in that region. Again, there is significant reduction in computational requirements and memory bandwidth by computing gradients only in the region around the keypoint instead of operating on the entire image. Once the score is computed, it is sorted to output the best N keypoints and the image level in which they were found. On these N keypoints, the rBRIEF descriptors are computed.

The rBRIEF descriptor is based on the rotated BRIEF algorithm [8] [9]. The descriptor is computed by picking a 48x48 window around each keypoint. We first compute the orientation of the keypoint by computing the moments as in [3]. We use the table lookup to generate the moment mask. Before computing the descriptor, the image is smoothened to reduce the effect of noise. We apply the 5x5 smoothing function only within the 48x48 window around each keypoint. This further reduces computational requirements and memory bandwidth compared to applying 5x5 smoothing function on the entire image. We use the table lookup to generate the 256 pairs of source-destination pattern needed to compute the descriptor.

The performance of the sparse point Harris score computation is 39 cycles/keypoint on the EVE. The rBRIEF descriptor computation takes 2192 cycles/keypoint on the EVE. The rBRIEF descriptor cycle count is higher as it also includes the smoothing filter and the moment computation.

3 Experiments and Results

In this section, we provide performance details of the algorithm implementation on the EVE engine of TI. We also show that it is reliable by providing the results of repeatability test.

3.1 Processing time

For our experiments, we consider an image of size 400x400. We compute the ORB descriptors on 3 levels of the input image i.e 400x400, 200x200 and 100x100. The maximum number of keypoints that can be detected in each level of the image is 2048. We then output the best 500 keypoints across the 3 levels and compute the descriptors for those. The algorithm has been implemented on EVE which is running at 650MHz.

In Table 1, column 1 indicates the stage of the algorithm, column 2 indicates the processing time taken in *cycles/pixel* and column 3 indicates total time taken in *ms* to process the 3 levels of 400x400 input image. Column 2 not only includes the computation cycles, but also the direct memory access(DMA) cycles required to move data between memories and other system level overheads. The total time taken is around 5.5ms. This is atleast 3 times faster compared to previous implementations mentioned in [3] and [4].

3.2 Repeatability

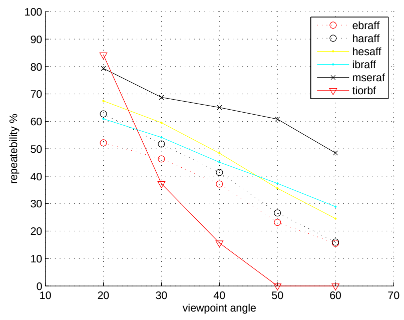
The detector is evaluated with the repeatability metric defined as the percentage of points simultaneously present in two images [10] [11]. The higher the repeatability rate between two images, better the correspondence between the keypoints

Table 1. Average processing time of various stages of ORB.

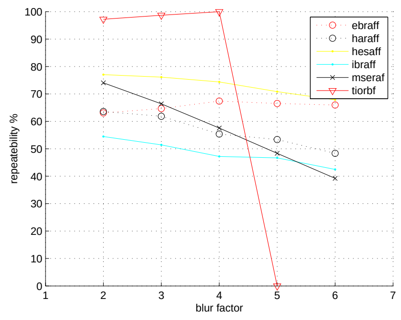
Stage	Processing time cycles/pix	Time taken in ms
Pyramid	0.43	0.138
FAST9	8.28	2.67
FAST9 score + NMS + Sort	181.8	0.572
Harris score + Sort	157.68	0.496
rBRIEF descriptor	2192	1.68
Total	2540.19	5.56

in the two images. We evaluate our detector for viewpoint changes and blurring. We use the images, Matlab code to carry out performance tests, and binaries of other detectors from http://www.robots.ox.ac.uk/vgg/research/afne for our evaluation.

Fig. 3a shows the repeatability rate of different detectors for viewpoint changes. Our implementation(tiorbf) outperforms other methods such as Edge based detector(ebraff), Intensity extrema based detector(ibraff), Maximally stable extremal regions(mseraf), Harris-Affine(haraff) and Hessian-Affine(hesaff) upto viewpoint changes of less than 30 degrees. Fig. 3b shows the repeatability rate of different detectors for blurred images. Blurred images were obtained by changing the focus of the camera. It can be seen that our implementation outperforms other methods for blur factors less than 5.



Viewpoint changes (a)



Blurring (b)

Fig. 3. Repeatability test viewpoint change and blurring.

4 Conclusion

In this paper, we have presented an efficient SIMD friendly implementation of ORB. We have provided key optimization techniques used in the implementation such as detecting 9 consecutive pixels, alternate non-iterative solution to compute FAST9 score and applying non-maximal suppression on sparse points. The main factor that contributes to the performance gain in our solution is that most of the processing is done on the sparse keypoints directly and not on every pixel of the image. We have shown that our solution is atleast 3 times faster compared to other approaches. Further, we have run the repeatability test to show that our solution is reliable despite the optimizations and modifications done.

One of the challenges that we have not addressed adequately is the approach taken during the non-maximal suppression. Our implementation can only perform 4-way non-maximal suppression, while the traditional approach uses 8-way non-maximal suppression. However, this can be addressed by performing an additional step post the 4-way non-maximal suppression to suppress based on neighbors along the diagonal direction.

References

1. Lowe, D.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision **60** (2004) 91–110
2. Bay, H., Tuytelaars, T., Gool, L.V.: Surf: Speeded up robust features. European Conference on Computer Vision **3951** (2006) 404–417
3. Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: Orb: An efficient alternative to sift or surf. Internation Conference on Computer Vision (2011) 2564–2571
4. Lee, K., Byun, K.: A hardware design of optimized orb algorithm with reduced hardware cost. Advanced Science and Technology Letters **43** (2013) 58–62
5. Lin, Z., Sankaran, J., Flanagan, T.: Empowering automotive with ti's vision accelerationpac (2013) http://www.ti.com/lit/wp/spry251/spry251.pdf.
6. Rosten, E., Porter, R., Drummond, T.: Faster and better: A machine learning approach to corner detection. IEEE Trans. Pattern Analysis and Machine Intelligence **32** (2010) 105–119
7. Rosten, E., Drummond, T.: Machine learning for high-speed corner detection. **1** (2006) 430–443
8. Huang, W., Wu, L.D., Song, H.C., Wei, Y.M.: Rbrief: a robust descriptor based on random binary comparisons. IET Computer Vision **7** (2013) 29–35
9. Calonder, M., Lepetit, V., Strecha, C., Fua, P.: Brief: Binary robust independent elementary features. European Conference on Computer Vision **6314** (2010) 778–792
10. Mikolajczyk, K., Tuytelaars, T., Schmid, C., A Zisserman, J.Matas, F.T., Gool, L.: A comparison of affine region detectors. International Journal of Computer Vision (2005) 43–72
11. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. IEEE Transactions on Pattern Analysis and Machine Intelligence **27** (2005)